TCP

Flow Control

- 16 bits of *advertised window size*: for flow control

TCP

Flow Control

- 16 bits of *advertised window size*: for flow control

TCP implements *flow control*, i.e., adjusting the rate of sending packets up or down to make best use of current conditions (a) in the network and (b) in the receiving host

TCP

Flow Control

- 16 bits of *advertised window size*: for flow control

TCP implements *flow control*, i.e., adjusting the rate of sending packets up or down to make best use of current conditions (a) in the network and (b) in the receiving host

The advertised window deals with (b)

TCP

Flow Control

- 16 bits of *advertised window size*: for flow control

TCP implements *flow control*, i.e., adjusting the rate of sending packets up or down to make best use of current conditions (a) in the network and (b) in the receiving host

The advertised window deals with (b)

The destination has only a limited amount of buffer memory it can store new segments in

TCP

Flow Control

- 16 bits of *advertised window size*: for flow control

TCP implements *flow control*, i.e., adjusting the rate of sending packets up or down to make best use of current conditions (a) in the network and (b) in the receiving host

The advertised window deals with (b)

The destination has only a limited amount of buffer memory it can store new segments in

If the application is not reading the data as fast as it arrives, the buffer will fill up

TCP

Flow Control

The window size is the amount of buffer the receiver has left: the receiver sends this value in each segment going back to the sender

TCP

Flow Control

The window size is the amount of buffer the receiver has left: the receiver sends this value in each segment going back to the sender

If the space left is very small, the sender can slow down sending until space in the receiver is freed up

TCP

Flow Control

A

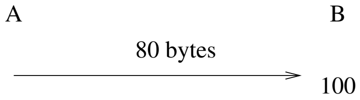
B

100

Initially B has space 100 in its buffer

TCP

Flow Control



A sends 80 bytes

TCP

Flow Control

A

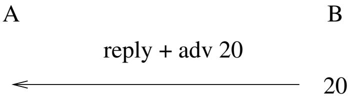
B

20

B save the data in its buffer

TCP

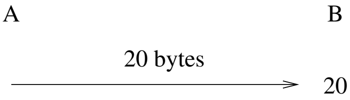
Flow Control



On the next returning segment, B advertises 20

TCP

Flow Control



A now knows it shouldn’t send more than 20

TCP

Flow Control

A

B

0

Next advertisement would be 0

TCP

Flow Control

A

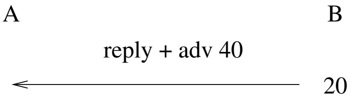
B

40

Until B reads some of the data

TCP

Flow Control



And can advertise the space

TCP

Flow Control

Thus B can tell A to slow down or speed up as appropriate to its remaining buffer space

TCP

Flow Control

Thus B can tell A to slow down or speed up as appropriate to its remaining buffer space

16 bits gives a maximum buffer of 65535 bytes: much too small for modern hosts that have megabytes to play with

TCP

Flow Control

Thus B can tell A to slow down or speed up as appropriate to its remaining buffer space

16 bits gives a maximum buffer of 65535 bytes: much too small for modern hosts that have megabytes to play with

There is a header option to scale this up to something reasonable

TCP

Flow Control

Thus B can tell A to slow down or speed up as appropriate to its remaining buffer space

16 bits gives a maximum buffer of 65535 bytes: much too small for modern hosts that have megabytes to play with

There is a header option to scale this up to something reasonable

Symmetrically, A has its own advertised window that it sends to B

TCP

Flow Control

Thus B can tell A to slow down or speed up as appropriate to its remaining buffer space

16 bits gives a maximum buffer of 65535 bytes: much too small for modern hosts that have megabytes to play with

There is a header option to scale this up to something reasonable

Symmetrically, A has its own advertised window that it sends to B

The other flow control mechanism to deal with varying conditions in the network comes later

TCP

- Checksum of the header, the data, *plus some fields of the IP layer*

TCP

- Checksum of the header, the data, *plus some fields of the IP layer*

Again, bad design!

TCP

- Checksum of the header, the data, *plus some fields of the IP layer*

Again, bad design!

- Urgent pointer: active if the URG flag is set

TCP

- Checksum of the header, the data, *plus some fields of the IP layer*

Again, bad design!

- Urgent pointer: active if the URG flag is set

The urgent pointer is a pointer into the data stream that indicates where the current *urgent data block* ends

TCP

- Checksum of the header, the data, *plus some fields of the IP layer*

Again, bad design!

- Urgent pointer: active if the URG flag is set

The urgent pointer is a pointer into the data stream that indicates where the current *urgent data block* ends

Urgent data includes things like interrupts that need to be processed before any other data that is buffered

TCP

The OS should notify the application when an URG is received, e.g., using an interrupt

TCP

The OS should notify the application when an URG is received, e.g., using an interrupt

The OS interrupt code would then read through the urgent data block and act appropriately on what it finds there

TCP

In a similar vein we have the

- PSH flag: set to indicate the destination OS should pass data to the application as soon as possible

TCP

In a similar vein we have the

- PSH flag: set to indicate the destination OS should pass data to the application as soon as possible

The destination OS might be holding back data for some reason before passing it on to the application, e.g., collecting together segments into one large buffer for efficiency reasons

TCP

In a similar vein we have the

- PSH flag: set to indicate the destination OS should pass data to the application as soon as possible

The destination OS might be holding back data for some reason before passing it on to the application, e.g., collecting together segments into one large buffer for efficiency reasons

Or holding back notifications to the application that data has arrived: again not to swamp the application with loads of notifications of small amounts of data

TCP

In a similar vein we have the

- PSH flag: set to indicate the destination OS should pass data to the application as soon as possible

The destination OS might be holding back data for some reason before passing it on to the application, e.g., collecting together segments into one large buffer for efficiency reasons

Or holding back notifications to the application that data has arrived: again not to swamp the application with loads of notifications of small amounts of data

This flag says send the buffered data to the application, don't wait

TCP

Originally it was intended the client application could set the PSH when it felt the server should not be hanging about buffering data

TCP

Originally it was intended the client application could set the PSH when it felt the server should not be hanging about buffering data

These days, there is no mechanism (in the sockets API) for applications to specify this, but the TCP software itself sets PSH when appropriate, e.g., when the client's send buffer empties

TCP

Originally it was intended the client application could set the PSH when it felt the server should not be hanging about buffering data

These days, there is no mechanism (in the sockets API) for applications to specify this, but the TCP software itself sets PSH when appropriate, e.g., when the client's send buffer empties

The idea here is that there is no point for the receiver waiting for more data, as there is no more to send right now

TCP

After the fixed header there are the options, including *window scale* and *maximum segment size*

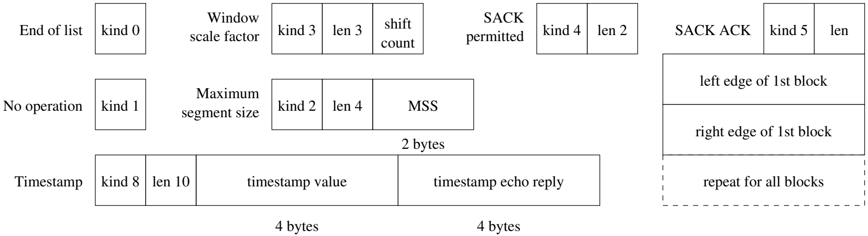
TCP

After the fixed header there are the options, including *window scale* and *maximum segment size*

After the options header is the data, which can be empty, e.g. for a pure ACK

TCP
Options

TCP Options are many and varied



Some TCP optional headers

TCP

Options

Options start with a 1 byte *kind* which indicates what the option is to do

TCP
Options

Options start with a 1 byte *kind* which indicates what the option is to do

Kinds 0 and 1 are one byte long; others have a length field

TCP

Options

Options start with a 1 byte *kind* which indicates what the option is to do

Kinds 0 and 1 are one byte long; others have a length field

No operation (NOP) is used to pad to align fields to a multiple of 4 bytes

TCP

Options

Options start with a 1 byte *kind* which indicates what the option is to do

Kinds 0 and 1 are one byte long; others have a length field

No operation (NOP) is used to pad to align fields to a multiple of 4 bytes

Maximum segment size (MSS) specifies how large a segment we can cope with: the headers are not included in count

TCP
MSS

The MSS is the largest TCP segment the host can process

TCP

MSS

The MSS is the largest TCP segment the host can process

Note that this segment *might* be reconstructed from more than one IP fragment, so might not be directly related to the MTU

TCP

MSS

The MSS is the largest TCP segment the host can process

Note that this segment *might* be reconstructed from more than one IP fragment, so might not be directly related to the MTU

However, if we want to ensure no IP fragmentation, the MSS must be set to the MTU minus headers: $40 = 20 + 20$ bytes for IP and TCP

TCP
MSS

The MSS is the largest TCP segment the host can process

Note that this segment *might* be reconstructed from more than one IP fragment, so might not be directly related to the MTU

However, if we want to ensure no IP fragmentation, the MSS must be set to the MTU minus headers: $40 = 20 + 20$ bytes for IP and TCP

Thus a TCP implementation must be able to process a MSS of $576 - 40 = 536$ bytes

TCP

MSS

The MSS is the largest TCP segment the host can process

Note that this segment *might* be reconstructed from more than one IP fragment, so might not be directly related to the MTU

However, if we want to ensure no IP fragmentation, the MSS must be set to the MTU minus headers: $40 = 20 + 20$ bytes for IP and TCP

Thus a TCP implementation must be able to process a MSS of $576 - 40 = 536$ bytes

The MSS is usually communicated in the option header in the setup of the TCP connection, and is typically set to avoid fragmentation

TCP

Options

As previously mentioned, the *window scale* option allows us to multiply up the value in the advertised window size header field

TCP

Options

As previously mentioned, the *window scale* option allows us to multiply up the value in the advertised window size header field

This optional field contains a value from 0 to 14

TCP
Options

As previously mentioned, the *window scale* option allows us to multiply up the value in the advertised window size header field

This optional field contains a value from 0 to 14

A value of n scales by 2^n: thus a maximum window of $2^{14} \times 65535 = 1,073,725,440$ bytes (a gigabyte)

TCP
Options

As previously mentioned, the *window scale* option allows us to multiply up the value in the advertised window size header field

This optional field contains a value from 0 to 14

A value of n scales by 2^n: thus a maximum window of $2^{14} \times 65535 = 1,073,725,440$ bytes (a gigabyte)

But that's still only about a second's worth of data in a 10Gb/s Ethernet!

TCP
Options

As previously mentioned, the *window scale* option allows us to multiply up the value in the advertised window size header field

This optional field contains a value from 0 to 14

A value of n scales by 2^n: thus a maximum window of $2^{14} \times 65535 = 1,073,725,440$ bytes (a gigabyte)

But that's still only about a second's worth of data in a 10Gb/s Ethernet!

A large window is very important is modern fast networks to get the most out of the available bandwidth: we don't want the client to have to keep stopping to wait for the server

My desktop uses a window scale of 7: $2^7 \times 65535 = 8388480$ bytes, or a maximum of 8MB buffer space per connection

TCP

Options

My desktop uses a window scale of 7: $2^7 \times 65535 = 8388480$ bytes, or a maximum of 8MB buffer space per connection

Its initial window size on a new TCP connection is 14600, meaning $2^7 \times 14600 = 1868800$ bytes, so a buffer of a bit under 2MB has been allocated (for this socket)

My desktop uses a window scale of 7: $2^7 \times 65535 = 8388480$ bytes, or a maximum of 8MB buffer space per connection

Its initial window size on a new TCP connection is 14600, meaning $2^7 \times 14600 = 1868800$ bytes, so a buffer of a bit under 2MB has been allocated (for this socket)

Exercise Go back and re-read the section on advertised windows

TCP

Options

Timestamp (TS val) puts the time of day into the segment header, allowing accurate measurement of the *round trip time* (RTT) of a segment and its ACK. Useful for computing retransmission times (see later)

TCP

Options

Timestamp (TS val) puts the time of day into the segment header, allowing accurate measurement of the *round trip time* (RTT) of a segment and its ACK. Useful for computing retransmission times (see later)

Timestamp Echo Reply (TS ECR) in an ACK segment is the timestamp being returned to the sender so it can compute the RTT

TCP

Options

Timestamp (TS val) puts the time of day into the segment header, allowing accurate measurement of the *round trip time* (RTT) of a segment and its ACK. Useful for computing retransmission times (see later)

Timestamp Echo Reply (TS ECR) in an ACK segment is the timestamp being returned to the sender so it can compute the RTT

Selective acknowledgement (SACK) is an extension of the ACK mechanism that allows more flexible ways of acknowledging segments. SACK is negotiated in the connection setup with a *SACK Permitted* option

TCP
Options

Several options are only allowed in the first segment of a new connection, e.g., Window scale, MSS and SACK Permitted

TCP

Options

Several options are only allowed in the first segment of a new connection, e.g., Window scale, MSS and SACK Permitted

This is because some things, e.g., buffer space, need to be set up before a connection and varying them mid-connection is difficult or makes little sense

TCP

Setup and Teardown

TCP is *connection oriented*, meaning a connection is set up between source and destination, and all packets that flow within this connection are related, through the sequence numbers, and they all use the same state, such as advertised window

TCP

Setup and Teardown

TCP is *connection oriented*, meaning a connection is set up between source and destination, and all packets that flow within this connection are related, through the sequence numbers, and they all use the same state, such as advertised window

For example, a connection to fetch a web page from a server will involve many segments

TCP

Setup and Teardown

TCP is *connection oriented*, meaning a connection is set up between source and destination, and all packets that flow within this connection are related, through the sequence numbers, and they all use the same state, such as advertised window

For example, a connection to fetch a web page from a server will involve many segments

Note that *each* TCP connection is separate from all others and has its own state

TCP

Setup and Teardown

TCP is *connection oriented*, meaning a connection is set up between source and destination, and all packets that flow within this connection are related, through the sequence numbers, and they all use the same state, such as advertised window

For example, a connection to fetch a web page from a server will involve many segments

Note that *each* TCP connection is separate from all others and has its own state

It is important to realise that this is a connection in the *transport layer*

TCP

Setup and Teardown

The underlying layer, IP, is not connection oriented, and each individual datagram is treated individually, e.g., might take a different route to its destination: IP is *connectionless*

TCP

Setup and Teardown

The underlying layer, IP, is not connection oriented, and each individual datagram is treated individually, e.g., might take a different route to its destination: IP is *connectionless*

Thus TCP connection has a weak version of sessions: though no further session mechanism is provided, e.g., no session resumption

TCP

Connection(less)

UDP is not connection oriented. Each datagram in UDP is treated individually

TCP

Connection(less)

UDP is not connection oriented. Each datagram in UDP is treated individually

UDP is a *connectionless* protocol

TCP

Connection(less)

UDP is not connection oriented. Each datagram in UDP is treated individually

UDP is a *connectionless* protocol

Of course, both connection oriented and connectionless protocols are useful in the right circumstances

TCP

Setup and Teardown

Setting up a TCP connection is complicated, as there is a lot of state that must be set up, e.g., sequence numbers, initial advertised windows, and buffers amongst other things

TCP

Setup and Teardown

Setting up a TCP connection is complicated, as there is a lot of state that must be set up, e.g., sequence numbers, initial advertised windows, and buffers amongst other things

Similarly, closing a connection is not trivial: we must ensure all segments in flight have been ACKed properly. Perhaps segments need to be resent. Thus a connection will hang around for a little after closing to ensure everything is tidied up

TCP

Setup and Teardown

Setting up a TCP connection is complicated, as there is a lot of state that must be set up, e.g., sequence numbers, initial advertised windows, and buffers amongst other things

Similarly, closing a connection is not trivial: we must ensure all segments in flight have been ACKed properly. Perhaps segments need to be resent. Thus a connection will hang around for a little after closing to ensure everything is tidied up

Fortunately for the application programmer, all this detail is taken care of by the TCP layer software in the operating system: though it does have occasional repercussions in the application if the connection needs to outlive the application for a while

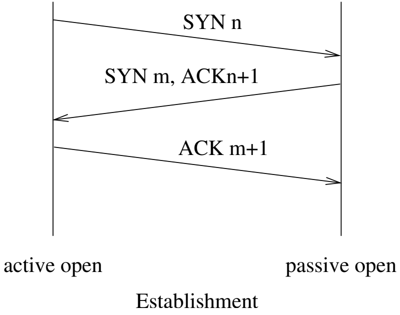
TCP

Setup and Teardown

Before TCP can send data, it exchanges some packets with the setup information

TCP

Setup and Teardown

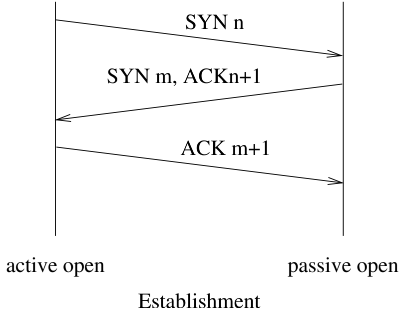


TCP setup handshake

Three segments are needed to exchange the information needed to make a new connection;

TCP

Setup and Teardown

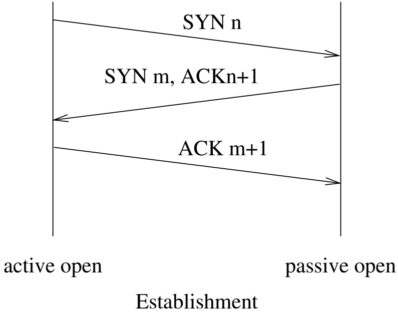


TCP setup handshake

The initiator, the *client*, sends a segment with the SYN flag set and its *initial sequence number* (ISN), *n*, is randomly generated;

TCP

Setup and Teardown

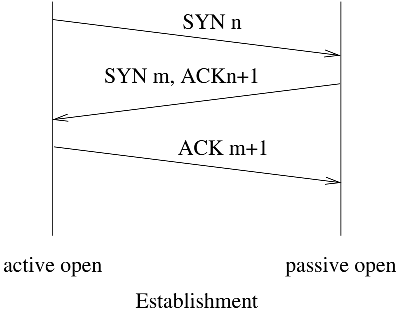


TCP setup handshake

The receiver, the *server*, replies with another SYN segment containing its own ISN, *m*;

TCP

Setup and Teardown

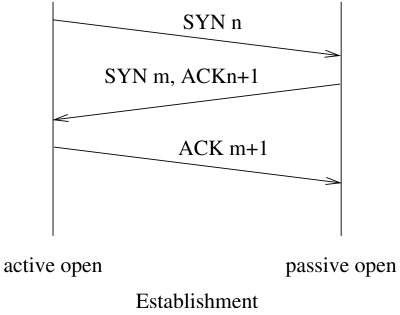


TCP setup handshake

It also ACKs the client's ISN with $n+1$, the sequence number of the next byte it expects from the client;

TCP

Setup and Teardown

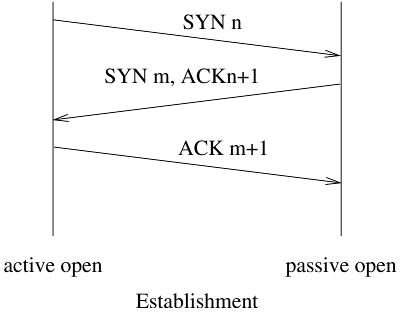


TCP setup handshake

The initial SYN can be lost just like any other segment, so we need to ACK it independently of the first data byte, which comes later;

TCP

Setup and Teardown



TCP setup handshake

The client ACKs the server's ISN with $m + 1$

This is called a *three way handshake*

TCP

Setup and Teardown

This is called a *three way handshake*

These segments contain no user data: they are overhead in setting up the connection

This is called a *three way handshake*

These segments contain no user data: they are overhead in setting up the connection

Overhead in time and overhead in packets on the network

TCP

Setup and Teardown

This is called a *three way handshake*

These segments contain no user data: they are overhead in setting up the connection

Overhead in time and overhead in packets on the network

After the handshake we can start sending data

TCP

Setup and Teardown

This is called a *three way handshake*

These segments contain no user data: they are overhead in setting up the connection

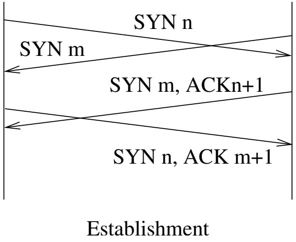
Overhead in time and overhead in packets on the network

After the handshake we can start sending data

The client (first one to initiate) is said to do an *active open*, while the server does a *passive open*

TCP

Setup and Teardown

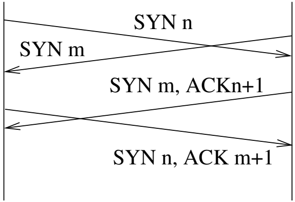


TCP simultaneous open

It is possible (but rare) for *both* hosts to do an active open, where the SYNs cross each other in flight

TCP

Setup and Teardown



Establishment

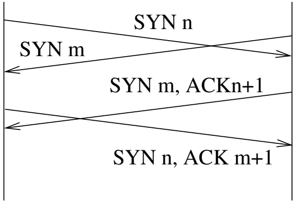
TCP simultaneous open

It is possible (but rare) for *both* hosts to do an active open, where the SYNs cross each other in flight

Matching TCP port numbers will identify when this happens

TCP

Setup and Teardown



Establishment

TCP simultaneous open

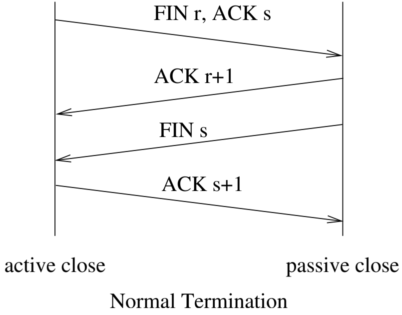
It is possible (but rare) for *both* hosts to do an active open, where the SYNs cross each other in flight

Matching TCP port numbers will identify when this happens

This is defined to produce *one* new connection, not two

TCP

Setup and Teardown

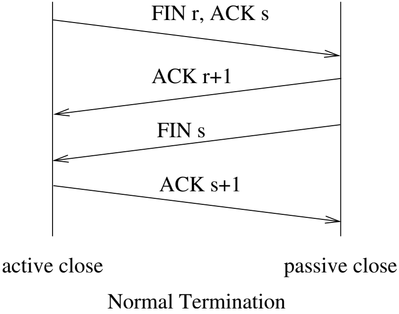


TCP teardown

Closing a connection takes up to four segments;

TCP

Setup and Teardown

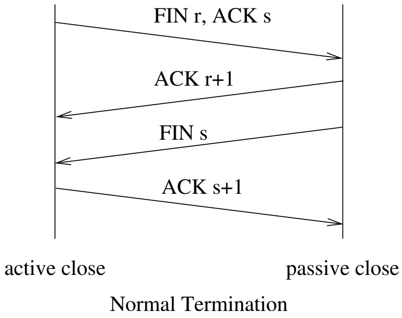


TCP teardown

TCP is full duplex, and a connection in one direction may be closed independently of the other;

TCP

Setup and Teardown

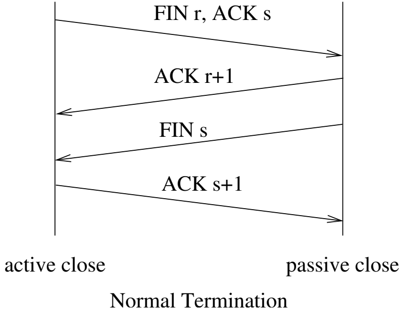


TCP teardown

The FIN flag is set to indicate a *half close*: this indicates no more data will be sent from this end;

TCP

Setup and Teardown

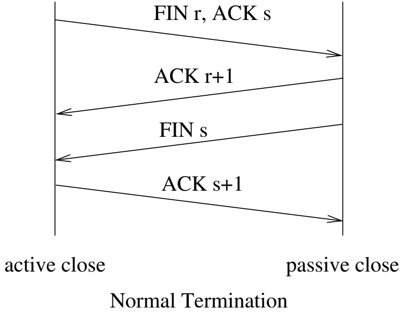


TCP teardown

We can still *receive* data at this end;

TCP

Setup and Teardown

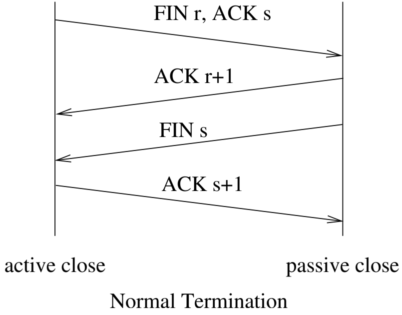


TCP teardown

The FIN is ACKed;

TCP

Setup and Teardown

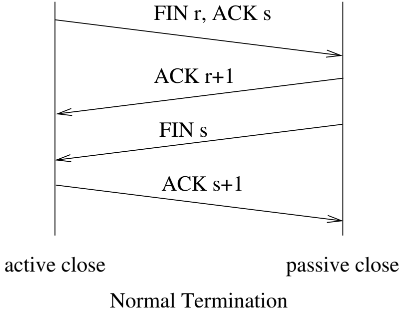


TCP teardown

When the other end wants to close, it sends a FIN and gets an appropriate ACK;

TCP

Setup and Teardown

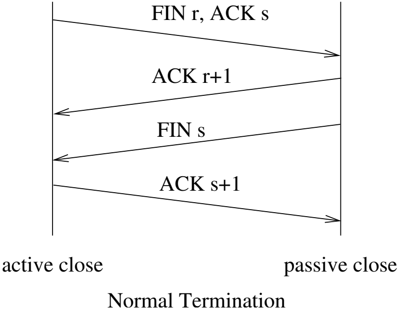


TCP teardown

Note there may still be data (and the corresponding returning ACKs) flowing from the server to the client before the server decides to close;

TCP

Setup and Teardown

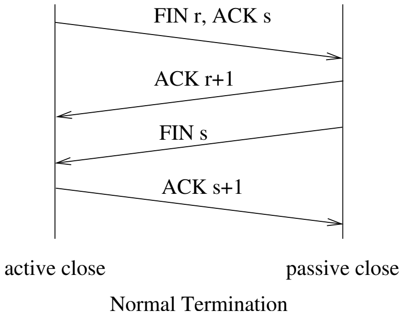


TCP teardown

The first close is called an *active close*;

TCP

Setup and Teardown

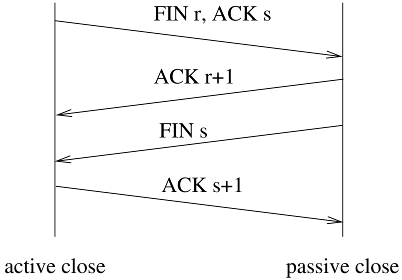


TCP teardown

The other end does a *passive close*

TCP

Setup and Teardown



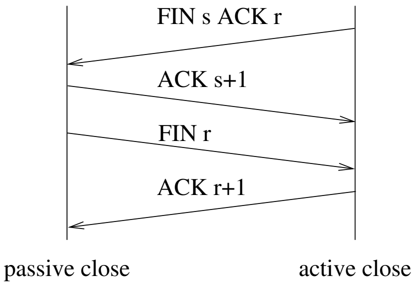
Normal Termination

Active close from left

Either end can initiate the active close; it does not need to be the host that did the active open

TCP

Setup and Teardown



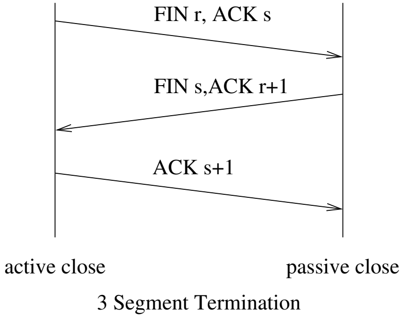
Normal Termination

Active close from right

Either end can initiate the active close; it does not need to be the host that did the active open

TCP

Setup and Teardown

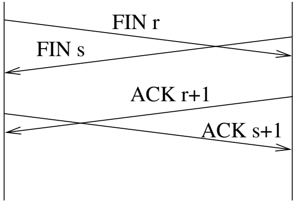


Three segment close

The passive close FIN can be piggybacked on the ACK: this then takes only three segments

TCP

Setup and Teardown



Termination

Simultaneous active close

There can (rarely) be a simultaneous active close: this takes four segments again

TCP

Termination

Connections are almost always ended by the FIN handshake, but there is another way to end a connection when something is badly wrong

TCP

Termination

Connections are almost always ended by the FIN handshake, but there is another way to end a connection when something is badly wrong

This is to send a *reset* (RST) segment, i.e., with the RST flag set

TCP

Termination

Connections are almost always ended by the FIN handshake, but there is another way to end a connection when something is badly wrong

This is to send a *reset* (RST) segment, i.e., with the RST flag set

This is for error cases, e.g., a segment arrives that doesn't appear to be for a current connection, the server will reply with a RST

TCP

Termination

Connections are almost always ended by the FIN handshake, but there is another way to end a connection when something is badly wrong

This is to send a *reset* (RST) segment, i.e., with the RST flag set

This is for error cases, e.g., a segment arrives that doesn't appear to be for a current connection, the server will reply with a RST

For example, if a server crashes and reboots while the client is still sending the server will not know what to do with the segments it is receiving; so it replies with a RST

TCP

Termination

When a host gets a RST it ends the connection immediately, discarding all state and buffered segments

TCP

Termination

When a host gets a RST it ends the connection immediately, discarding all state and buffered segments

Often seen by the application as a “connection reset by peer” message

TCP

Termination

A connection ended by FINs is called an *orderly release*; if ended by a RST it is an *abortive release*

TCP

Termination

A connection ended by FINs is called an *orderly release*; if ended by a RST it is an *abortive release*

RSTs are not ACKed: the connection ends right here

TCP

Termination

A connection ended by FINs is called an *orderly release*; if ended by a RST it is an *abortive release*

RSTs are not ACKed: the connection ends right here

Exercise Think about the security aspects of this: a third party can inject a RST segment into a connection to kill it